Introduction to (Thread) Synchronization

ICS332

Operating Systems

Introduction to Synchronization

- Synchronization is covered in depth in ICS432
- It's an important topic and it's difficult to do it justice in just a few lectures in an OS course
  - Although that's often done, sadly
- So we're only going to see the very basic concepts here, but get very little hands-on experience with synchronization
  - Take ICS432 if you do want such experience!!
- As a result, we'll only see a subset of the content in Chapter 5
  - Hence the very specific reading assignment

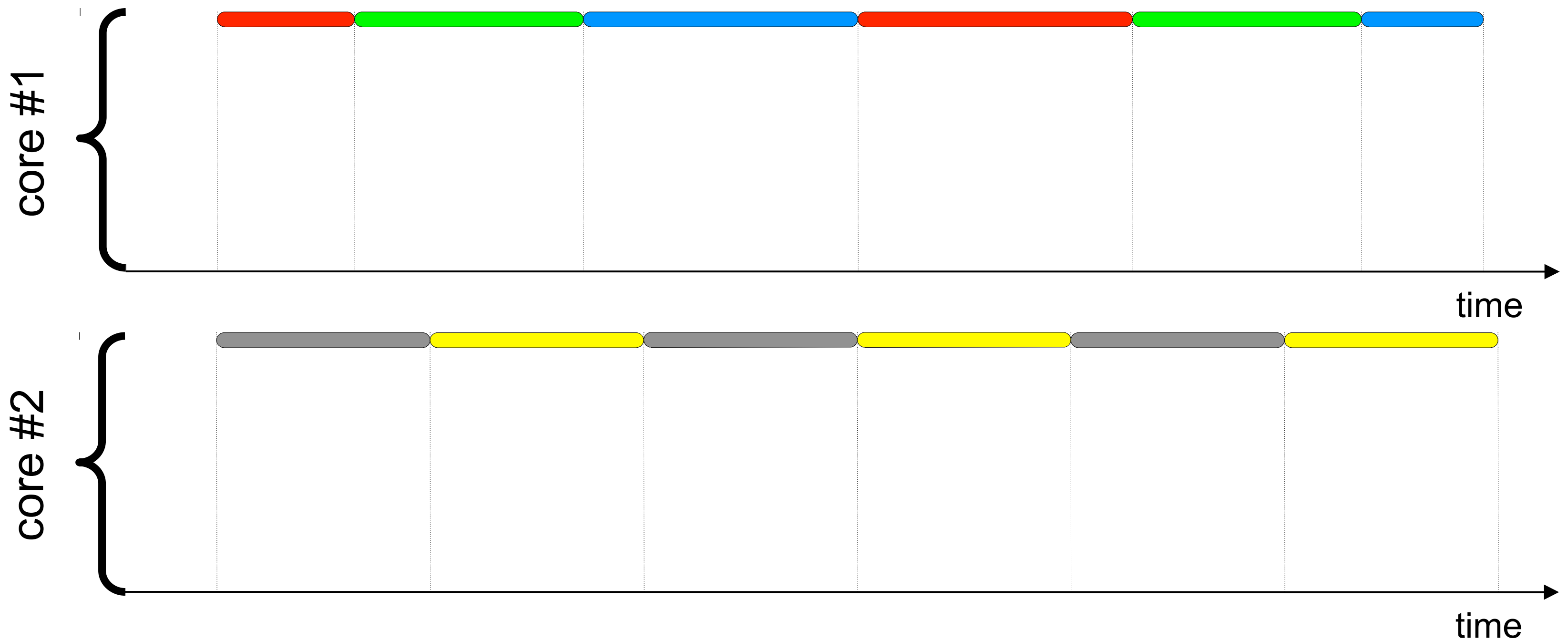
Cooperating Processes/Threads

- Having execution units run concurrently is useful
  - Structuring an application as independent but cooperating entities can be very convenient
  - Better utilization of hardware resources (e.g., cores)

- Different ways of doing concurrency
  - Multiple processes (with message-passing and/or shared memory)
  - Multiple threads in a single address space
  - All of the above together! (processes and threads are *tasks*)

Concurrency

- Two kinds of concurrency:



false concurrency within a core: illusion of concurrency provided by the OS
(e.g. green and blue task)
true concurrency across cores
(e.g., green and yellow task)

True/False Concurrency

- The programmer shouldn't have to care/know whether concurrency will be true or false
  - Typically, the programmer doesn't know on which core the program will run in the end!
- A concurrent program with 10 tasks should work on a single-core processor, a quad-core processor, a 32-core processor, etc.
- However, better performance with true concurrency
- We've talked about true concurrency across cores, but there could be true concurrency between any two hardware resources
  - e.g., between the network interface the core
  - e.g., between the disk and the network interface

Let's implement a... Counter

- Two Threads will access the Counter concurrently
- One will decrement the Counter value by 1 *n* times
- One will increment the Counter value by 1 *n* times

- Let's code it...
- Run for n=10, n=100, n=1000...
- Add debugging messages
- Run again for n=10, n=100, n=1000...

See Counter.java in ics332.rc.v1

Race Condition

- What we observed:
  - There **is** a **race condition**

    (i.e., the program is buggy)
  - The bug did manifest itself by several lost updates
  - It may not manifest itself, yet the program is still buggy

Concurrency Dangers

- There are two main problems with concurrent programs:
  - Race Conditions: a bug that leads the program to gives unpredictably incorrect results
    - Typical with processes/threads sharing memory
  - Deadlocks: the program blocks forever
    - Possible in any distributed system
- Let's first talk about Race Conditions
  - Arguably the most common/vexing problems
  - You will, unfortunately, encounter them
  - Deadlocks are in their own lectures notes

Why Race Conditions?

- Race conditions can happen with false or true concurrency
  - Statistically they're most likely to manifest themselves with true concurrency
- The *counter += increment* and *counter -= increment* statements are written in a high-level language
- The compiler translates them into machine code (or byte code if we are talking Java)... Let's have look at the assembly code
- On a Load/Store architecture (RISC), the code would then look like: *(check it yourself: gcc -S some_add_function.c)*

```
; Thread #1                ; Thread #2
load    R1, [@]            load    R1, [@]
inc  R1                    dec R1
store   [@], R1            store   [@], R1
```

Why Race Conditions?

- Illusion of concurrency: the OS context-switches threads rapidly
- We have 2 sets of 3 instructions, and thus many (?) possibilities
- Three possible execution paths

```
load R1, [@]
inc  R1
<# Context-switch #>
     loadR1, [@]
     dec R1
     store    [@], R1
<# Context-switch #>
store    [@], R1
```

```
load R1, [@]
inc  R1
<# Context-switch #>
     load     R1, [@]
     dec R1
<# Context-switch #>
store    [@], R1
     store    [@], R1
```

```
load     R1, [@]
<# Context-switch #>
     load     R1, [@]
     dec R1
<# Context-switch #>
inc  R1
<# Context-switch #>
     store    [@], R1
<# Context-switch #>
store    [@], R1
```

Important: R1 is not the same as **R1**
They are both register values into **logical** register sets (i.e., inside a data structure in the OS)

Why Race Conditions?

Let's assume that initially [@] = 5

```
load    R1, [@]   // R1 = 5
inc  R1           // R1 = 6
load    R1, [@]   // R1 = 5
dec R1            // R1 = 4
store   [@], R1   // [@] = 4
store   [@], R1   // [@] = 6
```

```
load    R1, [@]   // R1 = 5
load    R1, [@]   // R1 = 5
dec R1            // R1 = 4
inc  R1           // R1 = 6
store   [@], R1   // [@] = 4
store   [@], R1   // [@] = 6
```

```
load    R1, [@]   // R1 = 5
inc  R1           // R1 = 6
load    R1, [@]   // R1 = 5
dec R1            // R1 = 4
store   [@], R1   // [@] = 6
store   [@], R1   // [@] = 4
```

We would expect [@] to be 5 at the end
But we get 4 or 6

Lost Update

- In general, when a thread does “x++” and another does “x--” three things can happen
  - Both updates go through, the x is unchanged
  - The “x++” update is lost, and the value of x is decremented only
  - The “x--” update is lost, and the value of x is incremented only

Race Condition Example

- Assume we have two global variables a and b, initially both set to 1
- Thread #1:

```
a++;
b = a+2;
```

- Thread #2:

```
a--;
```

- Once both threads are finished, the main thread prints the value of a and b
- Question: what are the possible values?

```
a=1; b=1;
```

Thread #1

```
a++;
b = a + 2;
```

Thread #2

```
a--;
```

- First thing to do: come up with all possible interleaving of the instructions assuming that all instruction is executes entirely without being interrupted

```
a--;
a++;
b = a + 2;
```

```
a++;
a--;
b = a + 2;
```

```
a++;
b = a + 2;
a--;
```

```
a=1; b=1;
```

Thread #1

```
a++;
b = a + 2;
```

Thread #2

```
a--;
```

- First thing to do: come up with all possible interleaving of the instructions assuming that all instruction is executes entirely without being interrupted

```
a--;
a++;
b = a + 2;
```

```
a++;
a--;
b = a + 2;
```

```
a++;
b = a + 2;
a--;
```

a = 1, b = 3

a = 1, b = 3

a = 1, b = 4

```
a=1; b=1;
```

Thread #1

```
a++;
b = a + 2;
```

Thread #2

```
a--;
```

- Second thing to do: lost updates
  - Each line of code consists of multiple “hardware” instructions
- In this case: bad interaction between “a++” and “a--”
  - Result: a = 2
    - “a--” reads value 1, computes 0, gets interrupted
    - “a++” reads value 1, computes 2, gets interrupted
    - “a--” writes value 0
    - “a++” writes value 2, overwriting the 0
  - Result: a = 0
    - Same as “a=2” just different order
  - Result: a =1
    - Everything went well, without lost update
- We end up with two new possible output:

```
a = 0, b = 2
```

```
a = 2, b = 4
```

```
a=1; b=1;
```

Thread #1

```
a++;
b = a + 2;
```

Thread #2

```
a--;
```

```
a = 1, b = 3
```

```
a = 1, b = 3
```

```
a = 1, b = 4
```

- Output produced for all possible interleaving of lines of code
  - Can be considered a bug or not depending on what you application does
  - An application must not necessarily be 100% deterministic to be ~~correct~~ acceptable
    - Input could be random anyway

```
a = 0, b = 2
```

```
a = 2, b = 4
```

- Output produced due to the lost update problem
  - Typically considered a bug because a has a value different from 1 after “a++” and “a--” in the code, and b can take value 2 which likely makes no sense

Let’s try this program...

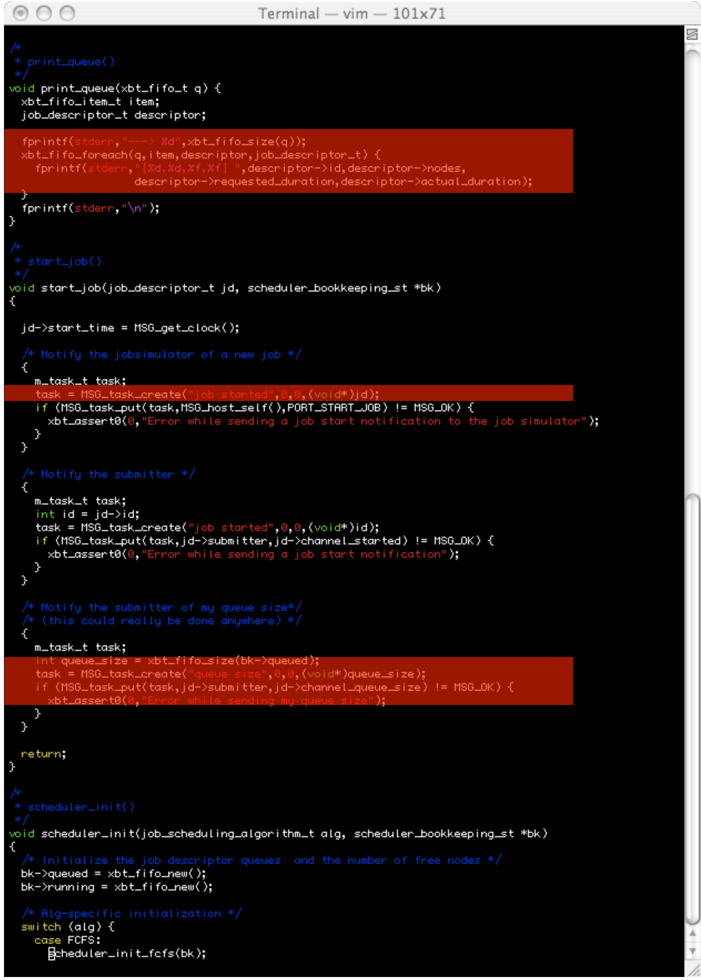
- RaceCondition2.java on the Web site
- Let’s run it 1000 times and see how many different outputs we get...
  - Let’s get this started and check back on it in a while....

Race Conditions Debugging = Nightmare

- A code may be working fine a million times, then fail once. Will it take one more million times to reproduce the failure?
- If you modify the code (e.g., adding a few print statements), or if you run in debugging mode, the race condition may no longer manifest itself or manifest itself more
  - The famous “I just added a print and everything works!”
- If you write code, run it, and it works, you don’t really know whether you’ve written a bug-free program
  - Typically true (even ith 100% coverage), but exacerbated with race conditions
  - You can prove a program wrong, but not a program right!
- **Non-deterministic** bugs are much harder to identify and fix
- So what can/do we do?

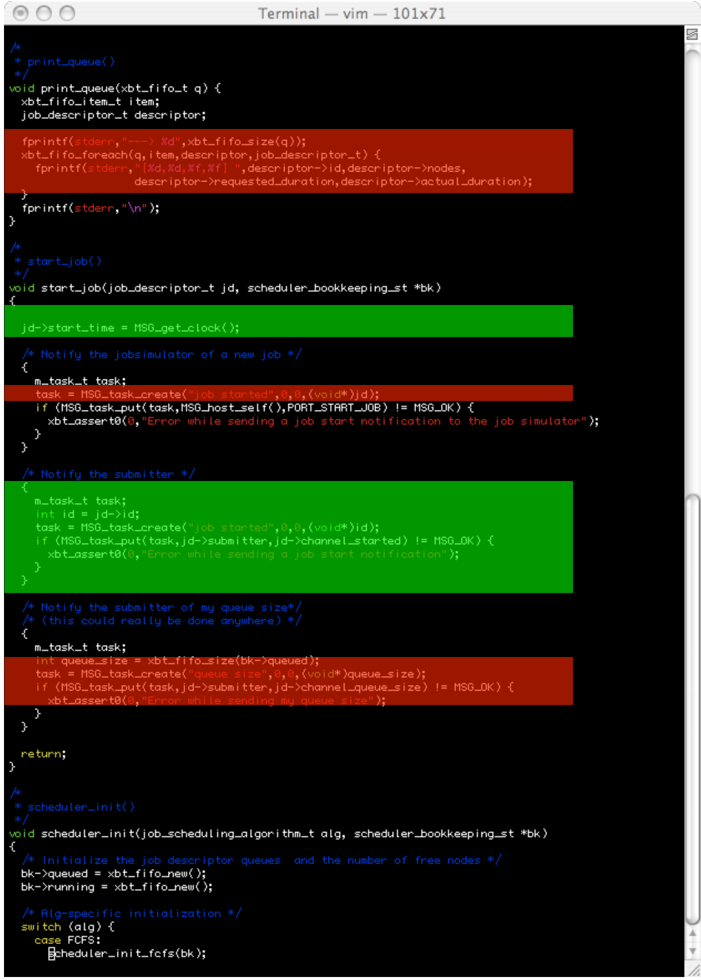
Critical Section

- A part of the source code where a race condition can happen is called a critical section
  - It doesn't have to be a contiguous section of code
  - In the example here, we have a 3-zone critical section
- For correctness only one thread can execute the code in a critical section at a time
  - If thread A is already executing one of the “red zones”, then all other threads must be blocked before being allowed to enter the same (or any) red zone
  - Only one will be allowed to enter once thread A leaves the red zone it was in



Critical Section

- We can have multiple critical sections
  - One 3-zone “red” critical section
  - One 2-zone “green” critical section
- In our initial example, we’d simply put the count++ and count-- statements in a (possibly multi-zone) critical section



Critical Section

- Formally, we want three properties of critical sections:
  - Mutual Exclusion: if thread T is in the critical section, then no other thread can be in it.
  - Progress: if thread T wants to enter into a critical section it will enter it some time in the future
  - Bounded Waiting: once thread T has declared intent to enter the critical section, there must be a bound on the number of threads that can enter the critical section before T
- Note that there is no assumption regarding the elapsed time spent by each involved process in the critical section

Critical Section: Common Misconception(s)

- A Critical Section corresponds to sections of code (i.e., the text segment)
- It doesn't correspond to data (i.e., variables)
  - Even though the section of code is typically one that modifies particular variables
- When we say "we need to protect variable x against race conditions" it means "we need to look at the entire code, see where x is modified, and put all those places in the SAME critical section"
  - If software engineering is well-done, modification of a single variable doesn't happen all over the code
  - And maybe now you see why global variables are "evil"
- It is a misconception that critical sections are attached to variables

From the OS point of view…

- What if a context-switch happens during a system call?
- Non-preemptive kernels ~~do~~ did not allow that
  - The thread runs until it willingly exists kernel mode (or yields control of the CPU)
  - No race condition!
  - Simple
- Preemptive kernels do allow a thread executing kernel code (in kernel mode) to be preempted
  - There can be race conditions
  - More powerful
  - Better for “real-time” programming as a “real-time” thread can preempt a thread running in kernel mode
  - Should be more responsive for the same reason
- Modern kernels are preemptive

Critical Sections and the Kernel

- On modern OSes, multiple threads can be in the kernel
  - User Threads that are doing a system call and are in kernel mode
  - System Threads doing useful system things
- Therefore, the kernel is subject to race conditions
  - We’ve seen that kernel debugging is hard, that race condition debugging is hard, so we don’t want race conditions in the kernel
- Example: the kernel maintains many data structures
  - e.g., the list of open files
    - The list must be updated each time a file is opened or closed
    - This is very much like the Counter example
  - e.g., the list of memory allocations
  - e.g., the list of processes
  - e.g., the list of interrupt handlers
- The Kernel developer must avoid all race conditions for access to these data structures

Synchronization Implementation

- What we need is a way to implement *enter_critical_section()* and *leave_critical_section()* to **lock** and **unlock** the access to the critical section
- There are some pure-software "solutions" (mostly historical)
  - They can be very complicated
  - They're not guaranteed to work on modern architecture
  - See Section 6.3 in the book if interested
- What we need is help from the hardware to provide **atomic** (non-interruptible elementary) instruction(s)
- Wait! What about disabling all interrupts?
  - If you allow whatever user process to disable interrupts, what tells you it will enable them afterwards?
  - What if interrupts are needed for other purposes, such as a bunch of timers?
- Conclusion: although inside the kernel one could disable interrupts for specific purposes, one cannot use this mechanism in general

Atomic Instructions and Locks

- Modern processors offer **atomic instructions**
  - Instructions that are uninterruptible from issue to completion
- With atomic instructions it is easy to implement the “lock” abstraction
- A lock is an abstract data type with two methods: lock() and unlock()
  - To “acquire” and “release” the lock
- A critical section is defined as the segments of code in between pairs of lock/unlock calls for a given lock
- Example

```
Lock mutex = new Lock(); // mutex = MUTual EXclusion
...
mutex.lock();
 // All code here is part of the critical section defined by mutex
mutex.unlock();
```

Short Critical Sections

- Critical sections should be as short as possible
  - Not in lines of code, but in time to run these lines
- Long critical sections: only one thread can do work for a while, so we have reduced parallelism
  - Extreme situation: the whole code is critical
  - Not a good idea in the case of multiple cores
- Goal: Many small and short critical sections (with different locks)
  - Many threads can do useful work simultaneously

What do Locks do?

- Two kinds of lock implementations
- **Spin lock**: The thread constantly checks whether the lock is available in a while loop
  - Prevents others (e.g., unrelated) threads from using CPU cycles
    - Can be a big problem on a single-core system
  - Wastes power and dissipates heat
  - But the thread will acquire the lock "as soon as" it is released
  - Very little overhead as no kernel involvement
- **Blocking lock**: The thread asks the OS to be put in the Waiting/Blocked state and the OS will make the thread Ready whenever the lock has been released by another thread
  - Has higher overhead as system calls and running kernel code is involved, (minimizing locking/unlocking overhead is important)
  - But it does not waste CPU cycles by "spinning"

Spin vs. Blocking Lock

- Spinlocks are very useful for (short) critical sections
  - Burn only a few cycles, but provide fast response time because they do not involve the kernel
  - If your critical section is “x++”, definitely use a spinlock, not a blocking lock
  - Spin locks are used inside the kernel for speed
- Most kernels provide a blocking lock abstraction
  - To be used for long(er) critical sections

Thread Synchronization?

- It may be tempting to use locks for having two threads communicate
  - Thread A waits for an “event” by doing lock(x);
  - Thread B signals the “event” by doing unlock(x);
- This is not a good idea, and a separate abstraction is needed
- This abstraction is called a condition variable
- It provides two mechanisms:
- wait(): Ask the kernel to be put in the Blocked state
- signal() and signal_all(): Unblock a (all) blocked thread(s)
  - i.e., tell the OS that that thread is runnable again
  - Does not mean that the thread calling signal() relinquishes the CPU immediately: it’s only about some threads changing state
- Conceptually, the kernel has a queue of blocked threads for each condition variable

Thread #1	Thread #2
...	...
cond.wait();	cond.signal();
...	...

Condition Variables and Locks

- If a thread acquires a lock, and then calls wait() on a condition variable, then it is blocked and nobody else can get the lock!
  - General rule: don't go to sleep while you're holding a resource that could let a bunch of people do useful work (i.e., a lock)
- To enforce this, a condition variable is associated with a lock, and wait() **temporarily** releases the lock
  - This is safe because while a thread sleeps, it's not doing anything at all
- Pseudo-code for wait:

```
void wait(cond_t condition, lock_t mutex) {
     unlock(mutex);
    <ask the OS to put me into the blocked state and to unblock me when
         the event "condition" is signaled>
    lock(mutex);
}
```

Classical Synchronization Problems

- To explain/understand synchronization, many typical problems are used
- Some are things you'll implement often
  - Producer-Consumer, Reader-Writer, Bank Account, ...
- Others are interesting metaphors
  - Dining philosophers, Barber shop, ...
- Some are surprisingly difficult and finding good solutions has occupied many computer scientists
  - Much more in ICS432
  - You can read some of the book's content if you want
    - But there is much more to it anyway

Back to the Counter example

- Let's make our Counter **thread-safe**

  java.util.concurrent.locks.Lock

  (No need for conditional variable here but they exist in Java)

  See Counter.java in ics332.rc.v2 for spin lock

  See Counter.java in ics332.rc.v3 for blocking lock

- Run again for n=10, n=100, n=1000...

Monitors

- Writing concurrent programs with locks and condition variables is very error prone
  - Typically, either you're implementing a version of one of the well-known problems, or you're introducing concurrency bugs
    - At least as a beginner concurrent programmer
  - And even though, the producer-consumer wasn't super easy either
- In the 70s, Hoare / Brinch-Hansen proposed the concept of a Monitor
- A monitor is an abstract data type representing a shared "resource"
  - e.g., a class/object
- It is a construct of a programming language
- Java implements monitors
  - You can implement Lock and CondVar with Java monitors, but few people do this and just use monitors directly

Monitors

- There is nothing magical here, we still need the two basic functionalities of mutual exclusion and waiting/signaling
- Monitors have the same "power" as other synchronization abstractions such as locks and condition variables
- But monitors constrain several aspects
  - Condition variables are not visible outside the monitor
    - They are hidden/encapsulated
    - One interacts with them via special monitor operations
  - Mutual exclusion is implicit
    - Monitor operations execute by definition in mutual exclusion
- These apparently innocuous properties make writing concurrent code less error-prone
  - The programmer shouldn't have to deal with lock, unlock, wait, and signal
- The book describes Monitors in Section 6.7 in detail
- Let's talk about how Java does synchronization with monitors (Section 6.8)

Synchronization in Java

- Unbeknownst to you, all Java objects you have used in your life have have a lock and a condition variable “hidden” inside of them
  - And implement lock- and condvar-like methods/capabilities
- To ensure mutual exclusion, a method can be declared as synchronized:
  - e.g., public synchronized void addItem(Item E)
- All synchronized methods in a class are executed in mutual exclusion
  - This is sometimes overkill or downright a hindrance, so one can also ensure mutual exclusion for a block of code or for a class
    - See ICS 432
- Every object implements wait(), notify(), and notifyAll()

Back to the Counter example Java-style

- Using **synchronized** methods

  See Counter.java in ics332.rc.v4
- Using **synchronized** statements (intrinsic locks)

  See Counter.java in ics332.rc.v5

- Run again for n=10, n=100, n=1000...

Back to the Counter example: ultimate Java-style

- Use **Atomic** objects

  See Counter.java in ics332.rc.v6

- Run again for n=10, n=100, n=1000...

- Check the java.util.concurrent packages

Priority Inversion

- Going back toward the OS, we have seen that processes/threads can have different priorities
- Let's just say that a higher priority process, if ready, always runs before a lower priority process (like in priority scheduling)
- Important: Processes, even if their code doesn't lead to synchronization problems, use data structures in the kernel that are themselves protected by, e.g., locks
  - Whether you see it or not, your programs do use locks, cond vars, semaphores, etc. when they run in kernel mode
- Let's say we have three processes: H > M > L
  - Resource R (e.g., a linked list in which elements are inserted/removed) is currently in use by process L
    - Process L holds a lock called mutex
  - Process H requires resource R
    - Process H is blocked on a lock(mutex)
  - But process M is running, preventing process L from running for a long time
  - So process L can never call unlock(mutex)
- Priority Inversion: Process M runs, and runs, while process H is stuck

Priority Inversion Solution

- Most OSes implement a priority inheritance mechanism
- A process that accesses a resource needed by a higher priority process inherits that process' priority temporarily
  - Complexifies the Kernel code quite a bit
- This solves the example seen in the previous slides
- Read Section 6.5.4 and the "Priority Inversion and the Mars Pathfinder" blurb
  - The program was real-time, so higher-priority processes had better run when they need to!
  - If priority inheritance hadn't been implemented in the kernel of the OS, the pathfinder would have failed

Semaphores

- A semaphore is a synchronization mechanism that combines locks and condition variables
- We won't talk about it in this course
  - Take ICS432
  - See Section 5.6 in the book if interested

Synchronization Concerns

- Race Condition
  - Inconsistent program state leading to error or incorrect execution
- Deadlock
  - No thread can make progress
- Starvation
  - Some threads don’t get access to the CPU even though they should
- Unfairness
  - Some threads don’t get access to the CPU enough compared to other threads
- Livelock (Take 432 or read the book)
  - Constant flip-flopping without any progress being made

Synchronization in Solaris

- Solaris provides:
  - adaptive mutexes
  - condition variables
  - semaphores
  - reader-writer locks
  - turnstiles
- Adaptive mutexes
  - looks at the state of the system and “decides” whether to spin or to block
  - e.g., if the lock is currently being held by a thread that’s blocked, forget spinning
  - No matter what, long critical sections should be protected by semaphores or cond. variables so that one is certain that there will be no spinning

Synchronization since Windows XP

- The Kernel uses spin locks for protection within the Kernel
  - Or interrupt-disabling on single-processor systems
- It ensures that a (kernel) thread holding a spin lock is never preempted
- For user-programs, Windows provides dispatcher objects
  - mutex locks
  - semaphores
  - event (a.k.a. condition variables)
  - timers (sends a signal() after a lapse of time)
- MemoryBarrier (prevents the CPU from reordering read-write instructions)
- Same concepts

Synchronization in Linux

- Locking in the Kernel: spin locks and semaphores
  - Spin locks protect only short code sections
  - On single-core machines, disables kernel preemption
    - Which is allowed only if the current thread does not hold any locks (the kernel counts locks held per thread)
  - (Non-spin) Semaphores used for longer sections of code
- Pthreads
  - (non spin) mutex locks
  - spin locks
  - condition variables
  - read-write locks
  - Semaphores
  - Futex (fast userspace mutex) (since 2.6, Dec. 2003)

Conclusion

- Synchronization is an essential topic
  - Theory is difficult
  - Practice is difficult
- The future may change this unfortunate situation
  - “New” “concurrent” languages (Erlang, uC++, Go...)
  - New ways to think about concurrent programming
  - Help from the compiler
  - Help from the hardware: transaction memories
- If you want to know more, take ICS432